AFRL-RI-RS-TR-2008-97
In-House Interim Technical Report
March 2008

USING YFILTER CONCEPTS FOR FAST BROKERING IN THE JOINT BATTLESPACE INFOSPHERE (JBI)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2008-97 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
DUANE A. GILMOUR, Chief Computing Technology Applications Branch	JAMES A. COLLINS, Deputy Chief Advanced Computing Division Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAR 2008	Interim	Jan 07 – Dec 07

4. TITLE AND SUBTITLE

USING YFILTER CONCEPTS FOR FAST BROKERING IN THE JOINT BATTLESPACE INFOSPHERE (JBI)

5a. CONTRACT NUMBER
In-House

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62702F

6. AUTHOR(S)

Justin M. Fiore, Lei Zhao and Vincent J. Mooney III

5d. PROJECT NUMBER
459T

5e. TASK NUMBER
XJ

5f. WORK UNIT NUMBER
BI

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
School of Electrical and Computing Engineering, College of Computing
Georgia Institute of Technology
Atlanta GA 30332

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RITB
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2008-97

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA# WPAFB 08-1135

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The use of YFilter concepts within the XML brokering task of the Joint Battlespace Infosphere (JBI) is described. Concepts from the YFilter were implemented and extended in the C++ language. The benefit of using YFilter concepts in the brokering task is to share as much processing between similar predicates as possible. The resulting reduction in brokering time can enable faster performance and greater scalability in the JBI. The work presented includes an implementation of YFilter concepts in C++, integration with the 100X JBI, and performance analyses.

15. SUBJECT TERMS
Information Management, Joint Battlespace Infosphere, JBI, publish-subscribe, XML Brokering, YFilter

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			George O. Ramseyer
U	U	U	UL	43	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

List of Figures

List of Tables

1. EXECUTIVE SUMMARY

This paper describes the use of YFilter concepts within the XML brokering task of the Joint Battlespace Infosphere (JBI). XML Brokering in the context of the JBI is the determination of whether or not a publication matches the desired filters (predicates) specified by subscribers. The YFilter is a system designed for the XML brokering task. The YFilter aims to share as much processing between predicates as possible. This will reduce the total time required to evaluate all predicates for a given Information Object (IO). Concepts from the YFilter were implemented and extended in the C++ language. The resulting reduction in brokering time can enable greater scalability in the JBI. The work presented includes an implementation of YFilter concepts in C++, integration with the 100X JBI, and performance analyses.

The YFilter Broker was integrated with the 100X JBI and is currently the default broker. The YFilter Broker was tested on the AFRL Coyote High-Performance Computing cluster. The improvements described in this paper yielded up to a 15 fold decrease in brokering latency and up to a 15 fold increase in system throughput when compared with the prior software broker.

2. Introduction

The Joint Battlespace Infosphere (JBI) is envisioned to be an information management system that allows users to dynamically provide, discover, and exchange information. Among the services of the JBI are the Publish and Subscribe services, which allow JBI clients to communicate via the publish-subscribe paradigm. Metadata in the JBI are specified in Extensible Markup Language (XML) [1], thus requiring the JBI infrastructure to perform XML brokering. This report presents an approach to improve the speed of XML brokering by implementing concepts introduced in the YFilter project [2] and [3]. The expected gain is the improved scalability of brokering latency and system throughput as the number of JBI subscribers continues to increase.

3. Background Information

3.1. Overview of JBI

This section provides a brief overview of the JBI. JBI terms are *italicized* in this section. In the JBI, an *Information Object* (*IO*) is the unit of information that is published, stored, and/or retrieved. An *Information Object* is of a specific type defined by an XML schema and consists of metadata in an XML document and a payload either in XML or in another supported format [7]. *Information Object* types are versioned. A JBI publication is sent from a publishing *JBI client* to a *JBI server*. A subscribing *JBI client* may request conditional delivery of publications by prescribing a predicate in the XML Path Language (XPath) [4]. The *JBI server* evaluates these predicates and transmits any matching publications to the subscriber. In this paper, we assume the reader has a working knowledge of JBI terms. For more information, the reader is directed to [5], [6], and [7].

The interactions of the *JBI Clients* attached to a particular *JBI Server* are described in the following general example.

> One or more subscribing *JBI clients* connect to a *JBI server* by issuing subscription requests. Each request specifies the *Information Object* type that the subscriber wishes to receive as well as XPath predicates on the metadata of the corresponding *Information Object* type. Upon accepting the subscriptions, the JBI Server commits to sending all publications matching the IO types and metadata to each subscriber. Publishing *JBI Clients* then connect by issuing a publication request. The publication request specifies the IO type that the *JBI Client* will publish. Upon accepting the publications, the publishing *JBI Clients* may publish *Information Objects* of the specified type. Those *Information Objects* will be brokered (the process of determining matching publications to subscriptions) and disseminated (distributed) to the matching subscribers.■

JBI Clients (both publishers and subscribers) can connect to the *JBI Server* at any time. The subscriptions received by subscribing *JBI Clients* are *Information Objects* that have been published after the subscription request is completed. In addition, the *JBI Server* only forwards *Information Objects* sent by publishing *JBI Clients* after the publication request has been completed.

3.2. Nondeterministic Finite Automata

A Nondeterministic Finite Automaton (NFA) is similar to a Deterministic Finite Automaton (DFA) in that an NFA is a collection of states and transitions where the transitions are taken while consuming input. Both NFAs and DFAs only recognize regular languages. The major difference is that the execution of an NFA is nondeterministic in the sense that for any given input symbol there may be more than one next state. In this sense, an executing NFA may be in multiple states at any given time. In addition, an NFA allows epsilon transitions, which are transitions that consume no input. The epsilon transition, in addition to the ability for a state to have multiple outgoing transitions with the same symbol, leads to the possibility to be in multiple states at any given time. The notion of acceptance in an NFA is such that if at the end of the input, any of the current possible states is an accepting state, the input is accepted. An NFA can be converted to a DFA and vice-versa. An NFA typically requires fewer states than a DFA to represent a given grammar. Because an NFA and a DFA can both represent the same grammar, the NFA will accept the same grammar of the corresponding DFA. In short, the output of the NFA used in this work is deterministic. For more information about Nondeterministic Finite Automata, please refer to [8].

3.3. Terminology

The following terms are used throughout this report:

Simple Predicate: The part of a *Clause* that contains value comparisons.

- E.g., the bracketed part in: /a/b/c[text() = “hello”]

Path Predicate: The part of a *Clause* that has bracketed nested *Location Steps*.

- E.g., the bracketed part in: /a/b/[./c/d/e]/f/g

Value-Based Predicate: A Simple Predicate or a Path Predicate.

Name Test: A string that identifies the name of an XML node. If the name test is '*', it will match an XML node with any name.

- E.g., the name test 'c'

Child Operator: The '/' operator. The child operator selects nodes that are children of the current context.

- E.g., '/a' will select all nodes whose node name is 'a' that are the direct child of the current context.

Descendant Operator: The '//' operator. The descendant operator selects nodes at the child level and all levels below.

- E.g., '//a' will select all nodes in the document whose node name is 'a'.

Location Step: A navigation axis (either / or //) followed by a *Name Test* and zero or more *Value-Based Predicates*.

- E.g., //c[text() = 4]

Clause or *Predicate Clause*: A set of *Location Steps*.

- E.g., /a/b//c[text() = 4]

Predicate: An XPath expression that consists of one or more *Clauses* combined by conjunctions (i.e., ‘and’, ‘or’, and ‘not’).

- E.g., /a/b//c[text() = 4] and /d/e/f[text() = 5]

Please note that we provide the above in order to have a quick reference handy for terms used commonly throughout this report. We have simplified the terminology for ease of discussion. For a more complete set of definitions, please see [7].

4. YFilter

This section discusses YFilter concepts in general and how they may be of use in the JBI context. The implementation of the YFilter concepts in C++ and their use in the JBI are discussed in later sections.

4.1. Overview of YFilter

The YFilter [2, 3] is a research project started at the University of California, Berkeley; IBM Almaden Research Center; and the University of Heidelberg. The YFilter aims to perform fast matching of XML documents against a large set of interest specifications and transformations of the matching XML documents based on interest specifications. The interest specifications are specified in a subset of XQuery [9]. The XQuery 1.0 specification is an extension of XPath 2.0. The filtering interest specifications, also called predicates, provide a Boolean match result for a given document and are specified using XPath expressions [4].

Since the JBI is most concerned with brokering an Information Object based on its XML metadata, we chose to restrict our implementation of the YFilter concepts to the matching of predicates and not consider the transformation of the output XML. According to the JBI specification, the subscription predicate of a subscribing JBI client either matches or does not match the IO's metadata [5]. Therefore, our implementation of the YFilter concepts consists only of Boolean matching of XML documents to predicates. Since we are not implementing the transformation part of the YFilter, we will not discuss transformation in detail. The interested reader can read more on this topic in [10].

In previous work, such as [11], it has been stated that filtering is the inverse problem of querying a database. In a database, a large dataset is stored persistently with queries coming in one at a time. In a filtering system, a large set of predicates is stored persistently with data items coming in one at a time. Therefore, while in a database, one would index (categorize) the data; in a filtering system, one should index (categorize) the predicates.

The YFilter compiles each predicate into a Nondeterministic Finite Automaton. The YFilter then combines all predicates into one shared NFA (Figure 1). This shared NFA takes advantage of any similarities in the predicate paths. In addition, a predicate can be added to or removed from the shared NFA by adding and removing states that are unique to the given predicate.

In Figure 1, two predicates that do not contain any value-based predicates are shown. The solidly colored circles are the states that are shared by the two predicates in the shared NFA. The other states are those that are not in common. The transitions between states are the name tests for each location step. This figure also demonstrates how descendant operators are converted into states. Each descendant has an epsilon transition to a state with a '*' transition and an outgoing transition of the name test that follows the descendant. Since the '*' transition matches any input, the descendant of the '*' transition introduces the requirement to potentially be in multiple states at any moment. The double circles are the accepting states for each of the predicates.

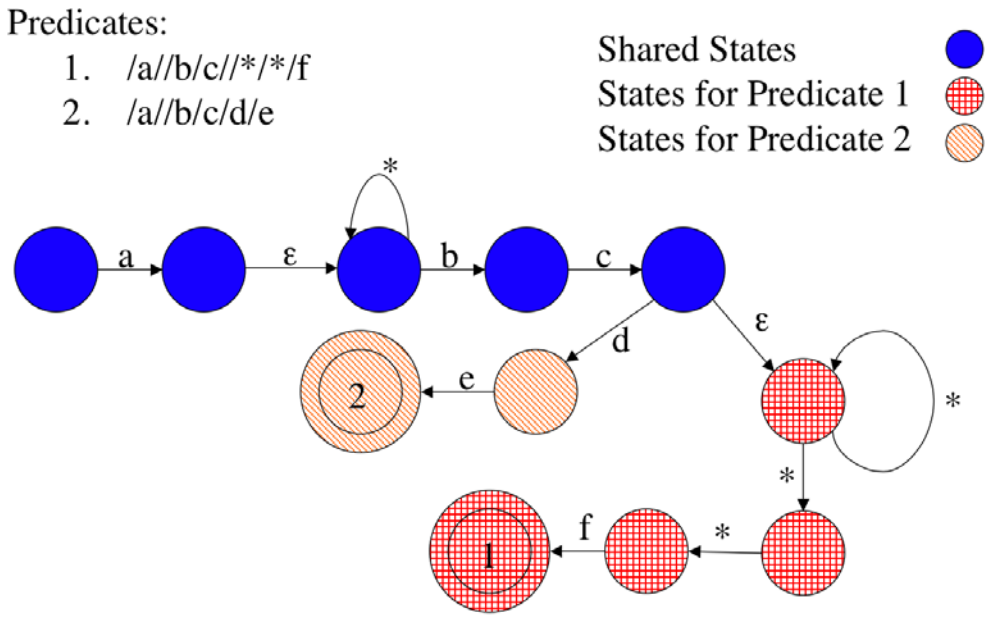

Figure 1: YFilter State Machine Example

As per the discussion of Nondeterministic Finite Automata in Section 3.2, the NFA in Figure 1 has an equivalent DFA. However, the number of states required to represent the descendant operator is far greater with a DFA.

The original YFilter is written in the Java language. Since the motivation of this work is to speed up the brokering for two JBI implementations, one in Java and one in C++, it made sense to implement the applicable concepts of the YFilter in C++ in order to achieve a native interface for the 100X JBI and to obtain a speedup from the C++ implementation.

4.2. Evaluation of Predicates

Once a set of predicates is compiled into a shared NFA, the NFA is driven by the parsing events produced by parsing an XML document. In particular, the events produced by the Simple API for XML Version 2.0 (SAX2) [12] are used to drive the transitions to the next states.

A predicate can consist of structural and *Simple* predicates. The structural predicate is the collection of *Location Steps* (including nested *Location Steps* in a *Path Predicate*) and evaluate to Boolean true only when those steps exist in an XML document. A *Simple Predicate* is the portion of the predicate that has a comparison of the text or attributes of an element to a value. The NFA only represents the structural portion of the predicates. The comparison of values is not expressed in states. In [2], the authors describe several options for evaluating the *Simple Predicate* portion of the predicate. They briefly describe treating the value comparison as states stating that this would increase the number of states and reduce the ability to share states. They then describe two approaches that they have implemented: *Inline* and *Selection Postponed*. Based on their report, we have chosen to implement the Selection Postponed method because their conclusion is that it is the better of the two. The interested reader can refer to the YFilter report [2] for a discussion of the Inline method. The Selection Postponed method waits until the state machine consisting of the structural portion is in an accepting state before performing any value comparisons. This requires some extra bookkeeping since we must be able to traverse the states previously visited if there are multiple *Simple Predicates* in a single clause. However, the

Selection Postponed method allows us to stop evaluation if any of the value comparisons in the predicate clause evaluate to false, thereby indicating that the overall predicate clause also evaluates to false (i.e., due to AND logic).

4.3. YFilter Grammar Limitations

The XPath Grammar supported by Diao's implementation of the YFilter is specified in the user manual that accompanies the source code [10]. The original YFilter implementation in Java has several limitations on the predicate grammar. In order to obtain optimal use of the YFilter concepts in the JBI context, extensions to this grammar needed to be made. The original YFilter grammar has the following limitations:

- Supports only one clause predicates.
- No support for AND, OR, and NOT operators (AND and OR are needed for multiple clause predicates).
- Value-based predicates only support string equality for attributes and element text.
- Does not support negative numbers.

These limitations are only engineering limitations, not theoretical limitations. As discussed in Section 5.4, we performed the requisite engineering work to extend the supported grammar.

4.4. References to "YFilter" in this Technical Report

Please note that, unless stated otherwise, all references to "YFilter" in this report are to the published and publicly available YFilter concepts. For example, if it turns out that there is a commercial company – e.g., a startup – producing proprietary YFilter implementations, nothing of the kind is in any way referred to or known about in this report (the authors of this technical report have no such knowledge of anything like a startup around the YFilter occurring at the time of writing this report). For a concrete example, the title of the next section – "YFilter C++ Implementation" – refers to C++ code implementing published, publicly available and non-proprietary YFilter concepts.

5. YFilter C++ Implementation

As part of this project, we implemented several of the YFilter concepts in the C++ language as a library that could be used by the JBI. We call this library "YFilter C++". The interface class to the library is NFAStateMachine. While a user of the library can create multiple NFAStateMachine instances, the instance's methods are not thread-safe. Therefore, in addition to the library, we also implemented a façade layer that allows thread-safe access. The façade also supports predicates that may not be supported by the YFilter C++ predicate grammar by evaluating these predicates without using the NFAStateMachine.

5.1. Overview of C++ Implementation

In this work, we focused on the brokering of Information Objects, which occurs in the publish/subscribe interactions of the JBI. The YFilter C++ work is probably not applicable to the query interactions of the JBI because, as stated in Section 4.1, database query is the inverse problem of filtering. Therefore, the YFilter solution to problem of filtering will not yield the best results for database query. We implemented the Selection Postponed method of simple value-based predicate evaluation (see Section 4.2 for a discussion of Selection Postponed).

5.2. Implementation Details

Figure 2 shows the architecture of the YFilter C++ library. This figure illustrates that both XPath predicates and XML documents enter the filtering engine and predicate matches for the documents exit the filtering engine. In the following discussion of Figure 2, the names of components are italicized.

The introduction of new predicates into the filtering engine starts with the predicate string entering the *XPath Parser*. The *XPath Parser* component consists of a scanner and parser generated from the Flex and Bison tools respectively. The *XPath Parser* is used to parse the XPath predicates so they can be compiled. In the *Predicate Compiler*, a predicate is compiled from its parse tree representation into an NFA without consideration of other existing predicates.

The *NFAStateMachine* then accepts these states and adds the unique states to the shared NFA that the *NFAStateMachine* maintains. For any common states, the *NFAStateMachine* registers the interest in those states of the predicate.

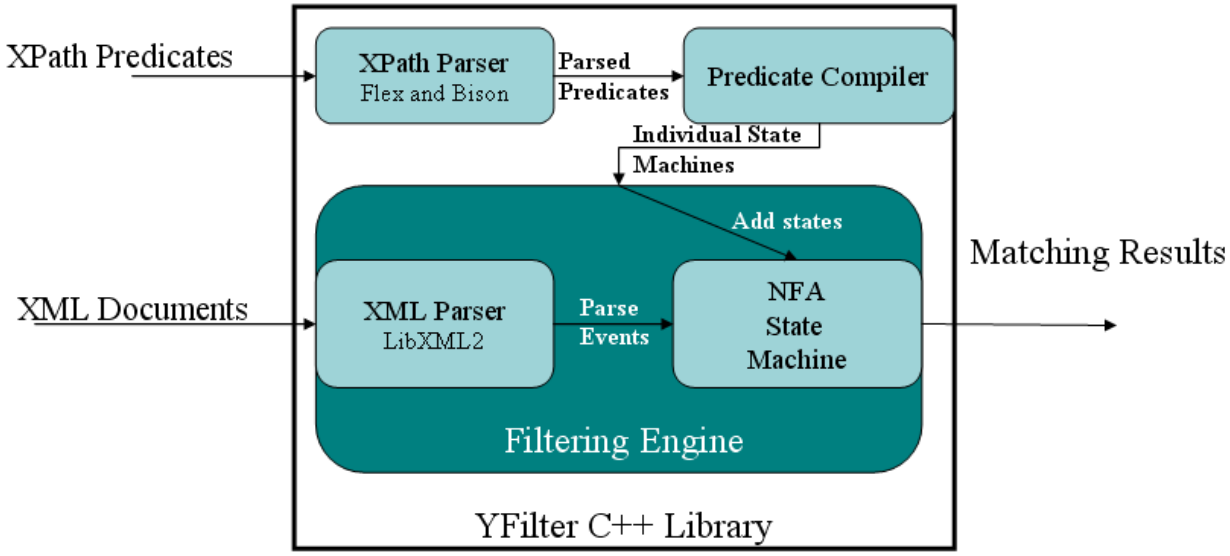

Figure 2: YFilter C++ Architecture

The other path shown in Figure 2 is the path taken when XML documents are evaluated. The XML string is consumed by the *XML Parser*, which uses the LibXML2 library. The *XML Parser* produces SAX events that drive the *NFAStateMachine*. The *NFAStateMachine* uses the SAX events to traverse transitions in the state machine. The *NFAStateMachine* then evaluates any simple value-based predicates for the accepting states in the state machine and evaluates the overall predicate by evaluating the AND, OR, or NOT operators if applicable. The *NFAStateMachine* then returns the set of matching predicate identifiers.

The YFilter C++ library is compiled into both dynamic and static libraries with the name libyfilter (libyfilter.a or libyfilter.so). In order to make the library as easy to use as possible, the interface that users of the library will use is the NFAStateMachine class. This class actually uses the other components in Figure 2 and isolates the user from interacting with the other

components directly. The interface uses the words “query” and “predicate” synonymously. In the context of the YFilter C++ library, both words refer to a *Predicate* as defined in Section 3.3.

The methods that the user of the YFilter C++ library will interface with are the *addQuery(string predicate)*, *removeQuery(XPQuery* compiledPredicate)*, *evaluateXML(string xml)*, and *evaluateXMLFile(string xmlFilename)* methods. The difference between the two evaluation methods is that the former parses XML contained in a string while the latter parses XML contained in a file. Before a predicate can be evaluated, it must be added to the state machine. This is performed by the *addQuery()* method, which parses the predicate, compiles it into an NFA, and integrates the NFA into the shared NFA. The *removeQuery()* method removes any states that are unique to the predicate being removed and cleans up any other data structures associated with that predicate. The evaluate methods evaluate all predicates existing at that time for the given XML document and return a result set containing the predicate identifiers for the predicates that successfully match the XML (i.e., yield a Boolean true).

The NFAStateMachine class implements the SAX2 parsing interface using the LibXML2 library. The LibXML2 library was chosen because it is available on almost every Linux distribution by default. In addition, the 100X JBI already uses this library for its brokering. It was decided that any predicates that cannot be evaluated using the YFilter C++ implementation (i.e., predicates not conforming to the YFilter C++ Predicate Grammar) would be evaluated using the current 100X method, which is to use LibXML2. In order to only have one parsing of the XML document, the NFAStateMachine class implements the LibXML2 SAX2 callback functions. The callback functions are called for various parsing events. The NFAStateMachine also builds up an XML Document Object Model (DOM) by calling the default LibXML2 handler functions so that when the SAX events are exhausted, the DOM may be used by any predicate evaluation that needs to be evaluated outside of YFilter C++.

The states and transitions that make up the state machine produced by the YFilter C++ implementation are equivalent to those produced by the YFilter Java implementation. However,

the data structures used to represent the state machine are different due to some mechanics of the C++ language and to accommodate integration with the LibXML2 XML parser [13].

5.3. Thread-Safety

The NFAStateMachine class, and therefore the libyfilter library, is not thread-safe. No two threads may call the *addQuery(), removeQuery(), evaluateXML(),* or *evaluateXMLFile()* methods simultaneously since they all either use or modify the underlying shared NFA. The façade layer has been created to present a thread-safe interface to the end-users of the library (see Section 6).

5.4. Extension of YFilter Predicate Grammar

The supported grammar for XPath predicates is documented in the *docs/supportedXPathGrammar.html* file provided with the YFilter C++ source code. The notable additions to the original YFilter predicate grammar have resulted in support for the following:

- Multiple clauses combined with AND, OR, and NOT
    - Includes any level of nesting
- Comparison operators less than, greater than, less than equals, greater than equals, and not equal
- Numerical comparison and equality (the YFilter Java implementation in [2] only implemented string comparison)
    - Supports doubles, longs and integers
- Negative numbers
- Shorthand for comparisons:
  E.g., /a/b/c = 5 is now valid and is equivalent to /a/b/c[text() = 5]
- Partial namespace support
    - The namespace prefix in the predicate must match the prefix used in the document

The XPath position() operator was removed because it has limited utility in the JBI context and would incur extra bookkeeping.

The type of the comparison operand is inferred from the XPath scanner generated by Flex. If the comparison operand conforms to the integer scanner token specification, the operand is interpreted as an integer. If the comparison operand conforms to the long scanner token specification, the operand is interpreted as a long. If the operand conforms to the double scanner token specification, the operand is interpreted as a double. If the comparison operand does not conform to the integer, long or double scanner token specifications, the operand is interpreted as a string. The value of the data in the XML document will be cast to the type inferred from the predicate. The following describes an example of this type inference.

> The predicate is '/a/b < 5.5'. The operand '5.5' will be treated as a double, and the text of the 'b' element in the XML document that is being evaluated will be interpreted as a double.■

With this extension, the YFilter C++ implementation can now perform numerical and string comparisons, as opposed to only string equality checking in the YFilter Java implementation.

Initially, YFilter C++ implemented the predicate grammar provided by Diao's YFilter [10]. The support of multiple clauses introduces some extra code. To support multiple clauses, each clause is compiled into an NFA. During the parsing, the values of the clauses are evaluated. After the document parse is complete, all predicate results are computed by performing any conjunctions based on the Boolean clause results. Clearly, adding support for multiple clauses increases the processing time for very complicated predicates that have many clauses and levels of nesting. However, without this, these predicates would not be optimized by the YFilter at all.

The benefit of the state machine evaluating the clauses and not the entire predicate is that more sharing can be obtained. In a large set of predicates, each with multiple clauses, there is a good chance that multiple predicates may contain clauses in common with each other. There is a lot of

efficiency gained when clauses are identical, because only one clause exists in the state machine, but the clause result can be used by all predicates that have that clause.

6. JBI Integration

6.1. Requirements

Both the JBI Reference Implementation (RI) and the 100X JBI benefit from use of the YFilter concepts to speed up brokering. The RI is written in Java [7] and the 100X JBI is written in C++ [5].

Both the RI and 100X JBI have the requirement that the predicate evaluator should not be limited by the supported predicate grammar of YFilter C++. Therefore, the interface should support all XPath expressions currently supported. Since the 100X JBI currently supports XPath 1.0 predicates, we chose the same compatibility level.

6.1.1. Reference Implementation Requirements

The RI had a PluggablePredicateEvaluator interface where predicate evaluators could be plugged into the RI. This interface had only one method shown in Figure 3 [14].

```
public interface PluggablePredicateEvaluator {
        public boolean evaluate(String predicate, String metadata);
}
```

Figure 3: PluggablePredicateEvaluator Interface

Unfortunately, this does not allow the specification of predicates ahead of time in order to compile a shared state machine. We worked with the JBI team to come up with a new interface that the broker utilizing the YFilter concepts could implement. This new interface is shown in Figure 4.

```
public interface BulkPredicateEvaluator {
        public void registerPredicate(String ioType, String ioVersion, String subscriptionUID, String predicate);

        public void unRegisterPredicate(String ioType, String ioVersion, String subscriptionUID);

        public void updatePredicate(String ioType, String ioVersion, String subscriptionUID, String predicate);

        public Vector evaluate(String ioType, String ioVersion, String metadata);
}
```

Figure 4: BulkPredicateEvaluator Interface

This interface allows the YFilter C++ library and any other bulk predicate evaluation system to plug into the Reference Implementation. In addition to implementing the methods above, the RI also required the interface to be thread-safe. The *updatePredicate()* method shown in Figure 4 can be used to change a predicate that is already registered. This was added to the interface for implementations that could update the predicate with better performance than simply unregistering the old predicate and registering the new predicate.

6.1.2. 100X JBI Requirements

The 100X JBI may have multiple disseminators, each of which is servicing multiple subscribers. A disseminator is the process that is responsible for distributing Information Objects to the subscribers. Each disseminator requires from the broker a list of network socket identifiers for which the publication matched. After evaluating the predicates, the broker must inform all disseminators which subscribers must receive the IO. Therefore, the original 100X JBI broker identifies each predicate by a disseminator identifier and a socket identifier. Both the disseminator ID and the socket ID are integer data types. The BulkPredicateEvaluator interface shown in Figure 4 specifies only one subscription user identifier (UID) value. The disseminator ID and socket ID could be converted into two strings, concatenated into a single string, and then used to identify predicates; however, this would incur the cost of parsing that string to obtain the numerical values for each predicate that successfully matched each document. This overhead was not desirable. Therefore, the disseminator ID and socket ID, each 32 bits in length, were combined into one 64-bit long long data type with the disseminator ID occupying the higher 32 bits and the socket ID occupying the lower 32 bits.

The 100X JBI can use multiple brokering processes, often running on different nodes of a High-Performance Computing (HPC) cluster. However, each brokering process must have all of the predicates as scheduling is performed in a round-robin fashion. In order to accomplish this, each brokering process must have its own instance of the NFAStateMachine. As currently implemented, the brokering process never adds a predicate, removes a predicate, or evaluates an XML document simultaneously. Therefore, the 100X JBI did not need thread-safe access to the YFilter C++ library.

6.2. Solution: YFilterPredicateEvaluator Template Class

The implemented solution to accommodate both the requirements of the RI and the requirements of the 100X JBI was to create a C++ template class [15] where the user of the template class could specify the type of the subscriptionUID. This solution would allow the RI to use strings and the 100X JBI to use long longs. This solution would also allow any other users to specify their own types. This template class would also provide a thread-safe interface to the underlying NFAStateMachine. The YFilterPredicateEvaluator template class implements this solution.

6.2.1. Managing Multiple NFAStateMachines

Since the JBI imposes a restriction that subscribers specify an IO type and IO version and that those subscribers will only receive publications of the specified type and version, the YFilterPredicateEvaluator uses a separate NFAStateMachine instance for each type and version combination. This not only helps avoid possible incorrect matches, but this also reduces the size of the state machines and minimizes the predicate evaluation to only those predicates which have successfully matched on type and version.

6.2.2. Evaluation of Incompatible Predicates

In addition, the YFilterPredicateEvaluator performs any predicate evaluation that is not supported by the YFilter C++ grammar. The YFilter C++ is first executed, which performs the parsing of the document and all evaluations that are YFilter C++ compatible. Then LibXML2 is

used to perform any remaining predicate evaluations using the DOM built by the YFilter execution. This ensures that any given document is only parsed once. This can be done because YFilter C++ implements the SAX2 handler of the LibXML2 library and forwards all handler calls to the default SAX2 handler to build the DOM prior to doing YFilter C++ computation.

6.2.3. Thread-Safety

The YFilterPredicateEvaluator also provides thread-safety. The YFilterPredicateEvaluator allows multithreaded access to different IO type and version combinations and synchronizes access to the same type and version. Also, for the same IO type and version, calls to *registerPredicate()*, *unRegisterPredicate()*, and *evaluate()* can occur simultaneously because *registerPredicate()* and *unRegisterPredicate()* do not actually modify the state machine. Instead, they perform as much of the processing as they can without the state machine and then enqueue the necessary modifications to the state machine to be performed later. However, they keep the expected semantics in that after *registerPredicate()* returns, matches on that predicate will be returned, even if the predicate is not integrated into the shared state machine yet. Likewise, when *unRegisterPredicate()* returns, matches with the corresponding subscriptionUID will no longer be returned. This delaying of addition and removal of states into the shared state machine was done to allow registration and unregistration without disrupting the critical path, which is predicate evaluation.

6.2.4. Modifying the NFAStateMachine

The YFilterPredicateEvaluator provides two ways to optimize the predicates and integrate them into the proper NFAStateMachine. In the following subsections, an alteration or modification of the NFAStateMachine is the addition or removal of a predicate (which may add or remove states within the NFAStateMachine).

6.2.5. The StateMachineAlterationThread

The first way to modify the NFAStateMachine is to use a background thread called the StateMachineAlterationThread. This thread is activated by calls to register and unregister. The StateMachineAlterationThread will then try to acquire locks for all state machines that need to

have modifications performed. If a lock is acquired, the needed modifications are performed, which is done by adding or removing predicates from the state machine. If the lock for a given state machine cannot be obtained, it is skipped and will be tried the next time the thread is activated. The following example demonstrates how the StateMachineAlterationThread works.

> For simplicity, IO versions will be omitted from this example. A predicate for IO type A is registered. The predicate is compiled into its individual state machine by the YFilterPredicateEvaluator, and the compiled predicate is added to the add queue. Then StateMachineAlterationThread (SMAT) is activated. It attempts to acquire the lock for the NFAStateMachine that is used for IO type A (SM-A). At this particular moment, SM-A is in the middle of an evaluation and the lock cannot be acquired.
>
> Sometime later, a predicate for IO type B is registered. The predicate is compiled and added to the add queue for the NFAStateMachine servicing IO type B (SM-B). The SMAT is activated again. The SMAT attempts to obtain the lock on SM-A and is successful. The SMAT adds the first predicate and any other compiled predicates in the add queue to SM-A. The SMAT then removes any predicates in the remove queue for SM-A. The SMAT then releases the lock on SM-A. Next the StateMachineAlterationThread attempts to obtain the lock on SM-B and is successful. The SMAT then processes the add queue and remove queue for SM-B. Lastly, the SMAT releases the lock on SM-B.■

This is the default behavior of the YFilterPredicateEvaluator and requires no further action by the user to optimize the predicates. However, the user can manually activate the StateMachineAlterationThread by calling the *updateStateMachine()* method. The obvious downside to using the StateMachineAlterationThread is that the optimization of the predicates and the overall performance of the brokering operation is less predictable and is partly based on how often the background thread can obtain the state machine locks.

6.2.6. Manually Requesting Modification of the NFAStateMachine

The second method of adding and removing predicates from the NFAStateMachine provides more predictable behavior but requires the user of the YFilterPredicateEvaluator to make explicit calls to the *updateStateMachine()* method with an optional maximum amount of alterations to complete. This allows the user to direct when and how many modifications will occur. This will also yield more predictable brokering latencies as the *evaluate()* method will not have trouble acquiring the state machine lock. This second option was added at the request of a 100X JBI team member who desired the ability to control the optimization and have more predictable

brokering latencies. The drawback of this option is that optimizations can be introduced sooner using the first method, especially during light evaluation load. At the time of writing this report, no decision has been reached as to which option is preferred for the RI.

6.3. Integration with the 100X JBI

The overhead of the synchronization in the YFilterPredicateEvaluator was not desired for the 100X JBI, which did not need the thread safety. Therefore, a separate concrete class (as opposed to a template class) called YFilterPredicateEvaluator100X was created with the same functionality, but with all of the synchronization code stripped out. This class is located in the 100X JBI code base in the PubCatcher subdirectory. The original YFilterPredicateEvaluator template class is still located in the YFilter C++ code base in the YFilterPredicateEvaluator subdirectory.

The JbiBroker class is the class that performs the brokering task in the original 100X JBI. In addition to a refactorization of the YFilterPredicateEvaluator for the 100X JBI, the YFilterBroker class was created. The YFilterBroker class inherits from the JbiBroker class, which was refactored to use virtual methods. This allows other implementations to extend the JbiBroker class. The YFilterBroker class conforms to the same interface as the JbiBroker and uses the YFilterPredicateEvaluator100X under the hood. The YFilterBroker class is also responsible for communicating with the disseminators for matching subscriptions. Since the YFilterBroker completely encapsulates the use of the YFilterPredicateEvaluator100X and conforms to the JbiBroker interface, the use of YFilter C++ has no impact on the rest of the 100X JBI code. By this measure, the YFilterBroker can be easily swapped out for the original JbiBroker or any other future broker that extends the JbiBroker class. The YFilterBroker is now enabled by default on the 100X JBI.

6.4. Integration with the Reference Implementation

The YFilterPredicateEvaluator template class conforms as closely to the BulkPredicateEvaluator Java interface as a C++ class can (given the differences between C++ and Java). There is a need

of a Java Native Interface (JNI) wrapper to translate between Java and C++. This JNI wrapper has not been created at the time this report was written. The implementation of the JNI wrapper should be straightforward because the only non-primitive items in the YFilterPredicateEvaluator template class interface are C++ STL classes. At the time of the writing, the integration with the Reference Implementation of the JBI is not complete.

7. Performance Analysis

7.1. Hypothesis

By the use of YFilter concepts in brokering we expect to see an increase in throughput and reduction of brokering latency as the number of subscriptions increase when compared to the previous 100X JBI broker (i.e., the JbiBroker class). Our measure of throughput is the number of Information Objects that can be pushed through the system per second (IOs / sec). Our measure of brokering latency is the amount of time it takes to evaluate all active predicates for a given message and to call the disseminators corresponding to the matching subscriptions. The set of active predicates is the predicates for a given set of subscribers. We do not expect that YFilter C++ will have a lower brokering latency for a single predicate, since there is extra overhead required in maintaining and executing the state machine. However, we do expect to see the YFilterBroker surpass the original 100X broker with only a few subscribers. The experiment in Section 7.3 will show that this point is at around 10 subscribers.

7.2. JBI Configuration

The 100X JBI can be configured in many ways, and can be distributed across multiple computing nodes. Unless otherwise specified, the configuration of the JBI for all of the experiments was that of Table 1. Three nodes of the Coyote High-Performance Computing (HPC) cluster were used. Each Coyote node has dual 3.06 GHz Intel Xeon processors, 4 GB DRAM and 400 GB disk space. The nodes were connected by a Gigabit Ethernet interconnection fabric [5]. Each node had a Subcatcher (i.e., disseminator) to distribute the dissemination load for large amounts of subscribers. In addition, each node had tasks associated with queries and archiving, though none of these tasks were invoked. All tests were strictly publish/subscribe tests and archiving was disabled.

Table 1: JBI Configuration

Node	Tasks
Coy13	Pubcatcher, Subcatcher, Connection Service, QueryCatcher
Coy14	QueryWorker, Broker, Archiver, Subcatcher, Archiver
Coy15	QueryWorker, Broker, Archiver, Subcatcher, Archiver

We used an Information Object that could obtain timing information internal to the 100X JBI server. The metadata was 604 bytes in length.

7.3. Broker Latency

The average broker latency was first measured. This is the mean time it takes to evaluate the existing predicates and call the appropriate disseminators with the matching results. Each data point is the mean of 9000 trials. The brokering latency was measured from 1 to 2900 subscribers. The maximum number of subscribers that was obtained was 2900 with the current 100X implementation due to factors (primarily in the operating system) unrelated to the YFilter C++ implementation. This was also the most subscribers that had been tested in any of the 100X JBI tests have been conducted with to date. When attempting to connect more than 2900 subscribers, the following error occurs on the 100X JBI server:

"*WARNING: ConnectionService: during first client receive: ChannelException: Failed on recv EFAULT.*"

For this experiment, we used two Coyote nodes besides those used for the 100X JBI server. One node published messages; the other node contained N subscribers, where N varied as high as 2900 subscribers. Each subscriber had the following two-clause predicate:

/message/from = 'RateTester' and /message/messno = $i

where the variable $i is replaced by the number of the subscriber. The publisher publishes messages where the /message/messno field circulates from 0 to N-1 and then wraps back around. This setup ensures that there is always one match. The publisher published messages as fast as possible. The fact that messages may become backed up in the pubcatcher queue is inconsequential because we are only measuring the brokering latency, not end-to-end latency. Also note that all data regarding broker latency was recorded using this configuration. The results for a different set of predicates will vary. The performance evaluation of a randomly generated set of predicates has not been tested yet. However, that may be performed in the future.

Measurements were taken after all predicates were optimized and integrated into the state machine. Measurements were taken for the original 100X JbiBroker and for the new YFilterBroker. Besides the switch of broker class, everything else was unchanged.

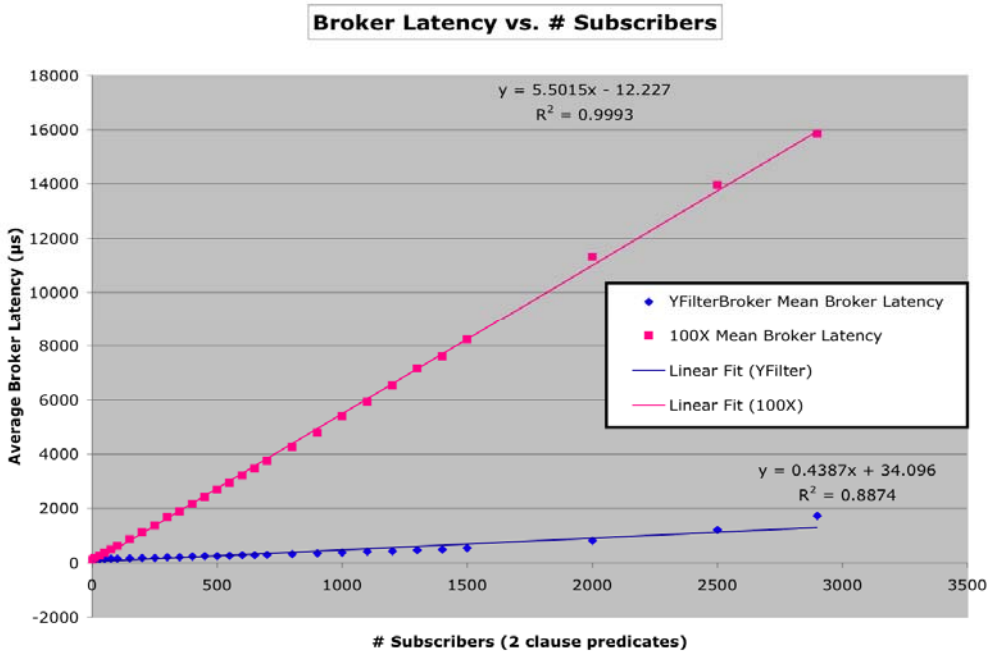

Figure 5: Broker Latency (µs) vs. # Subscribers

As shown in Figure 5, the YFilterBroker scales much better than the 100X JbiBroker. The intersection of the latency of the YFilterBroker and the 100X broker is after only a few subscribers. With 10 subscribers, the YFilterBroker evaluated faster than the 100X broker. The 100X broker crosses the 1 ms mark at about 200 subscribers, where the YFilterBroker crosses the 1 ms mark at about 2500 subscribers. The slope of the YFilterBroker is about 12 times smaller than the slope of the 100X broker. This means that the broker latency will scale about 12 times better with the YFilterBroker.

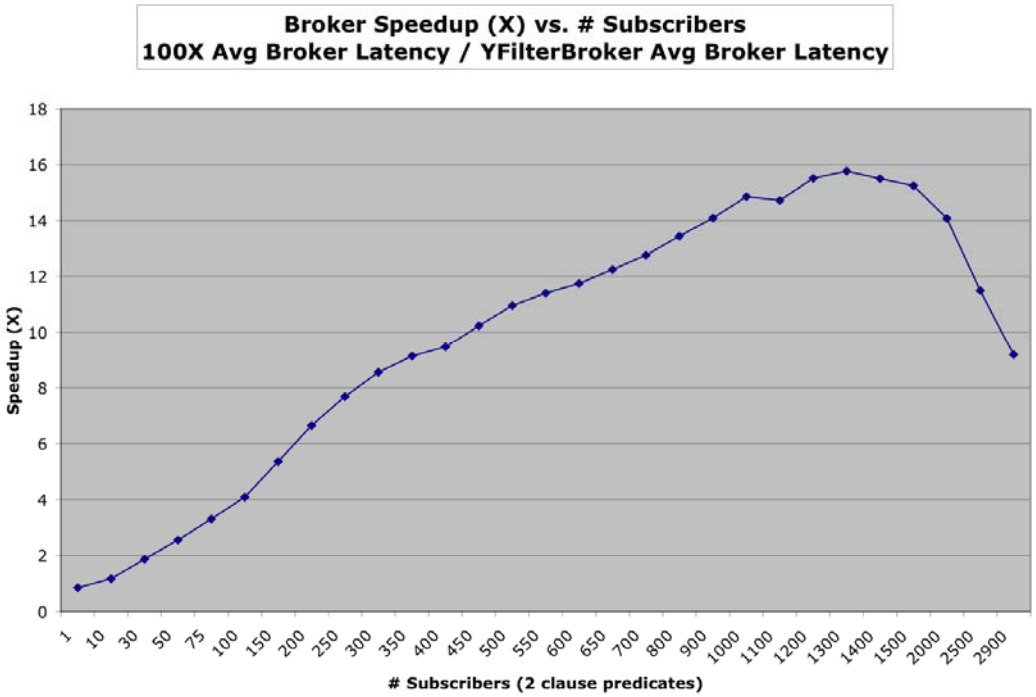

Figure 6: Brokering Speedup (X) vs. # Subscribers

The speedup (i.e., the ratio of average broker latency of the 100X broker over the average broker latency of the YFilterBroker) is shown in Figure 6. The speedup varies with the number of subscribers. However, we achieve a maximum speedup of about 15X. The speedup begins to decrease after a very large number of subscribers (about 1500). This is due to a slight upward curvature of average broker latency for the YFilterBroker (Figure 5) that starts to appear at about

1500 subscribers. We believe this to be due to an increase in cache misses with large numbers of subscribers. Because the evaluation of a predicate requires accessing one object for each clause and accessing one object for the predicate, each 2-clause predicate requires three object accesses for each subscription after the clause results are computed. Because each of these three objects must be accessed for each subscriber, we believe there to be an increase in cache misses during the processing required after clause results are computed. However, this belief has not been verified experimentally.

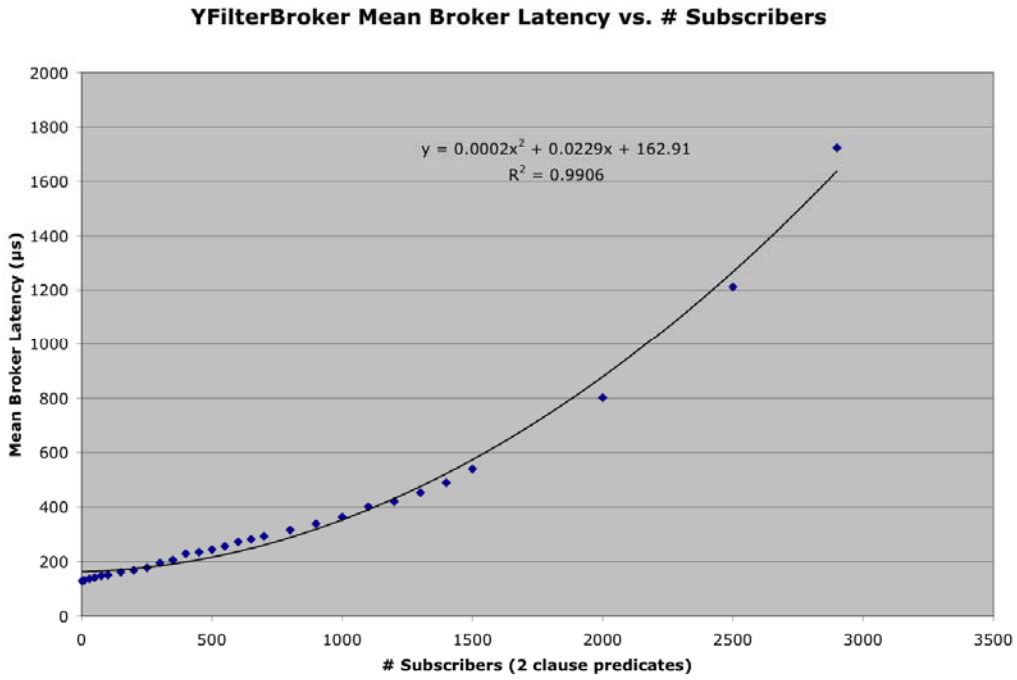

Figure 7: A closer look at YFilterBroker Broker Latency

Figure 7 is a graph of the broker latency using the YFilterBroker alone. We can see that the data appears to follow a quadratic curve rather than a linear line. This curve states that there is 0.2 ns * n^2 latency in addition to the linear term. The 0.2 ns coefficient is quite small, and if this trend continues, it will intersect the 100X JBI line again at 27,300 subscribers in this test case. Unfortunately, since 2900 is the maximum number of subscribers that can currently be simultaneously connected in the 100X JBI, we cannot run this test with more subscribers to see if

this trend continues. To this end, the next section describes a test that just measures the YFilter C++ evaluation time for up to 100,000 predicates.

7.4. YFilter C++ Evaluation Time

Due to the inability to test brokering latency above 2900 subscribers, we measured the YFilter C++ evaluation time up to 100,000 predicates using the same scheme for the predicates used in Section 7.3. The YFilter C++ evaluation time is a subset of the broker latency. The YFilter C++ evaluation time is strictly the time it takes for the YFilter C++ processing. This excludes the time spent in the YFilterPredicateEvaluator, which is negligible, and the time spent communicating with the disseminators. The YFilter C++ evaluation time includes all parsing and evaluation time due to the state machine.

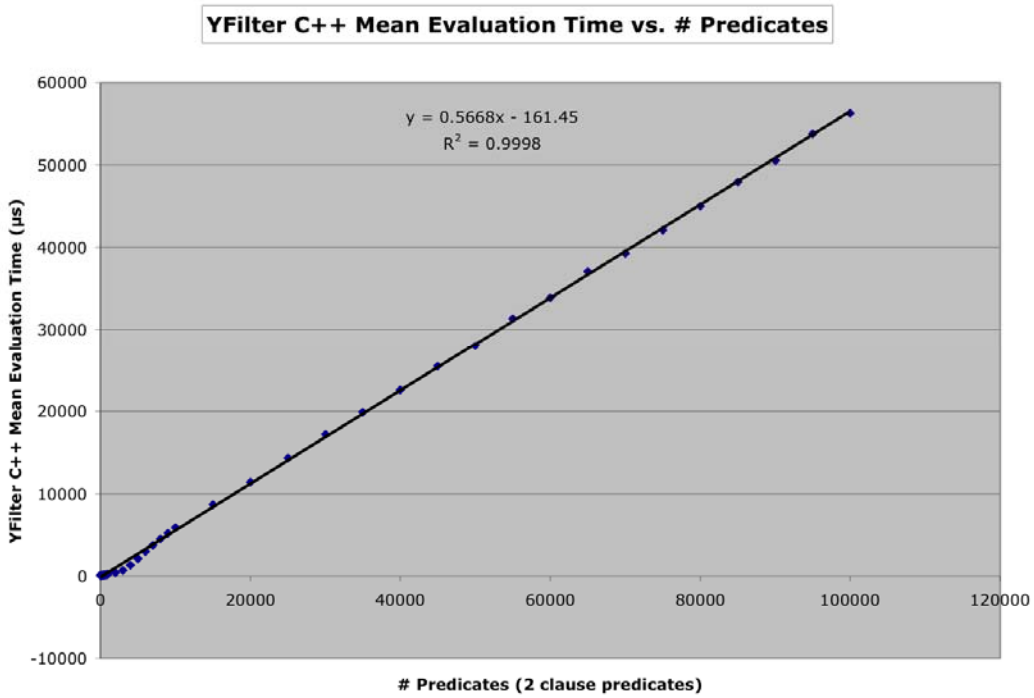

Figure 8: YFilter C++ Mean Evaluation Time (µs) vs. # Predicates

As shown in Figure 8, the YFilter C++ evaluation time is indeed linear, though it has a bit of a curve for lower numbers of predicates (e.g., less than 9,000 predicates). According to this data, the YFilterBroker should not ever intersect the 100X broker latency after the initial intersection

at around 10 subscribers. So even with the slight curvature for lower numbers of predicates, the YFilterBroker should still outperform the 100X broker even at large numbers of subscribers.

7.5. Throughput

The throughput in publications per second was measured for both the YFilterBroker and the 100X JbiBroker. The experiment configuration is shown in Figure 9. In this experiment, six nodes were used in addition to the nodes used for the JBI server. There were two publishers on each of three nodes. Each publisher published 10,000 publications of the testmessage IO as fast as possible. For each data point, N subscribers with predicates were subscribing. The other three nodes were used to run N/3 subscribers, where N varied up to 2900 subscribers. We used the same predicate scheme as Section 7.3. This ensured that only one subscriber would be sent each publication. This ensures that the bottleneck is the brokering task.

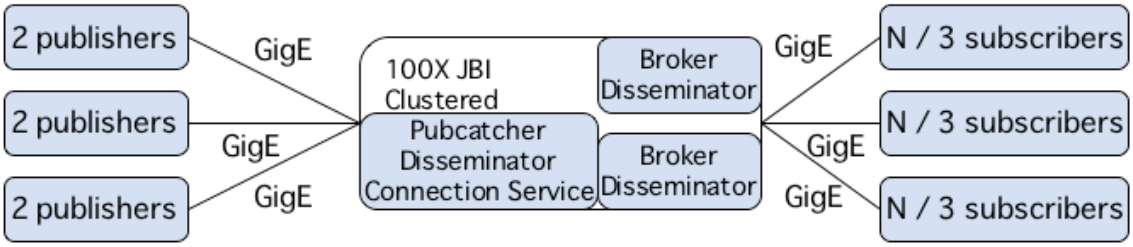

Figure 9: Throughput Experiment Configuration

Figure 10 shows the achieved throughput in publications/second for both the 100X broker and the YFilterBroker. From this figure, we can clearly see that the YFilterBroker scales better in terms of throughput than the 100X JbiBroker. The trend lines are shown to give a sense of how the throughput may continue to scale as more subscribers are added.

Figure 11 displays the benefit of the YFilterBroker in terms of the throughput ratio, which is the number of times more throughput the YFilterBroker can achieve over the 100X broker. With this experiment, we achieve a maximum of about 15X improvement over the 100X JBI. This is similar to the improvement gained in broker latency discussed in Section 7.3.

7.6. Performance Analysis Summary

The speedup obtained by the YFilterBroker will vary with the amount of similarity in the predicate set. However, for the experiments described in this section, the YFilterBroker is 15 times faster than the broker of the original 100X JBI. Since the 100X JBI is 347 times faster than the RI with only one broker [5], we expect to see even greater improvements when compared with the Reference Implementation of the JBI.

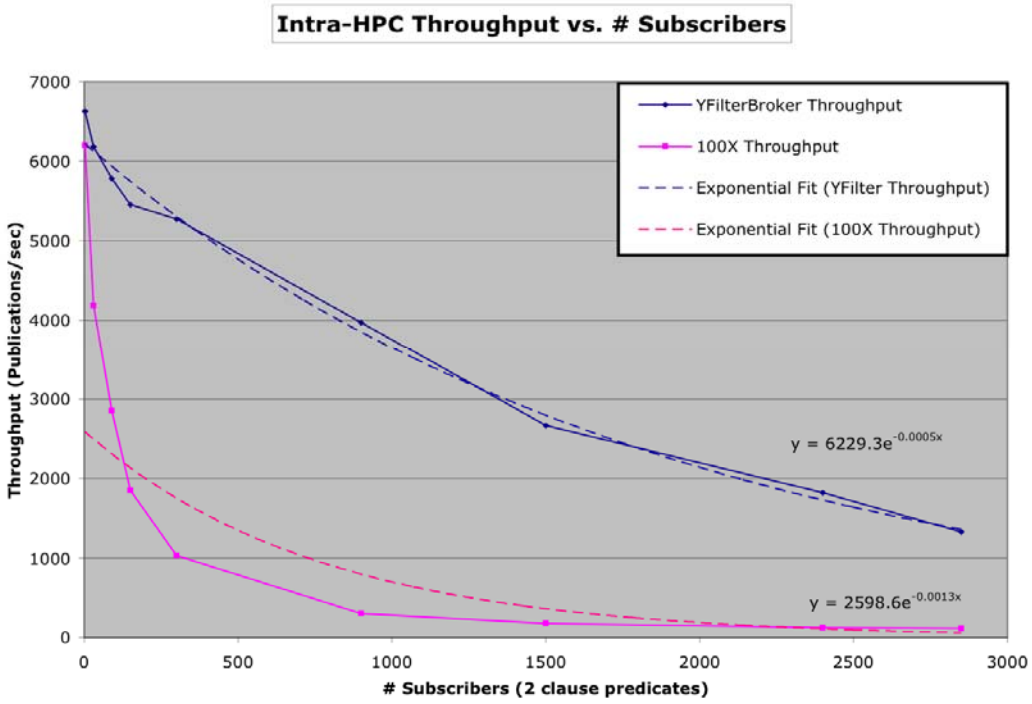

Figure 10: Throughput vs. # Subscribers

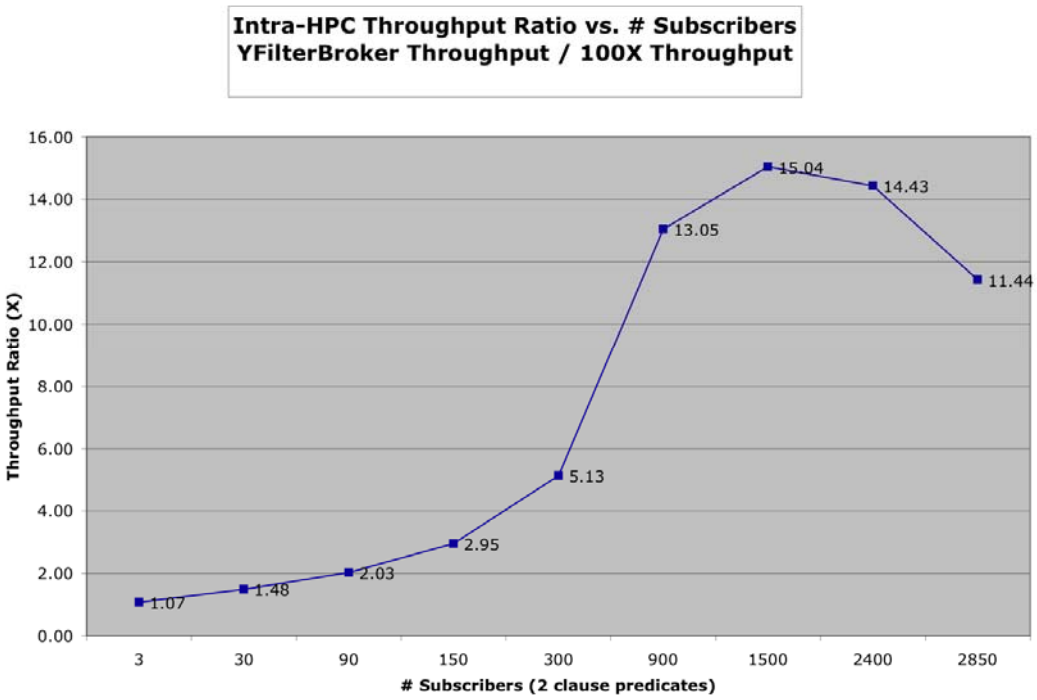

Figure 11: Throughput Ratio (X) vs. # Subscribers

8. Known Engineering Limitations

Beyond the limitations documented here, there are some minor future work items that are listed in the TODO.txt in the root directory of the YFilter C++ code base.

8.1. XPath Grammar Support

The XPath grammar supported by the YFilter C++ implementation only supports the child and descendant navigation axes. The current YFilter C++ code itself does not support other axes such as ancestor, parent, following, preceding, etc. However, the YFilterPredicateEvaluator brings the supported grammar up to the XPath 1.0 specification because of the use of the LibXML2 library as the fallback predicate evaluator. Some features of XPath 2.0 such as regular expression support may be desired by some versions of the JBI. Support for these features could be a future extension of this work.

The use of other navigation axes has not been thoroughly tested. There is one test case in the class NFAStateMachineTest that shows that other axes are properly rejected by the state machine. Further testing of how the YFilter appropriately rejects the navigation axes that are not supported and how LibXML2 handles them should be performed.

8.2. Type Inference of Value-Based Predicates

Currently, there is a limitation in that the type of the operand of the simple value-based predicate is inferred from the predicate. The data in the XML document being parsed is cast to the type inferred from the predicate. This could lead to a semantic misunderstanding. However, an easy user workaround is available.

Consider the example where a user supplies the following predicate: /a = 50. In this case, the type inferred would be an integer. If the document contained an element ‘a’ whose element text

was 50.1, this predicate would evaluate to true because the value in the document, 50.1, is cast to an integer prior to comparison.

It is easy to mitigate this mistake by simply changing the predicate to /a = 50.0 where the data is expected to be a floating point number. With this change, the inferred type is now a double, and the predicate will evaluate to false.

8.3. Sharing of Value-Based Predicates

In the current implementation, simple value-based predicates are only shared if there is an exact clause match in which case the entire clause is shared. This is not too detrimental because there are only two cases where two clauses that have the same path and same simple value-based predicate at the deepest level would not have an exact clause match. The first is when the clauses have simple value-based predicates that differ at other levels. For example, the two predicates:

```
/a/b[text() = 5]/c[text() = 10]
/a/b[text() = 6]/c[text() = 10]
```

would not share the evaluation of /c[text() = 10] because they do not have an exact clause match. However, from our present exposure to the types of predicates used in the JBI, this is not a common case. To date, we have not seen any predicates that have simple value-based predicates at multiple levels.

The second case is when two predicates differently use descendants. For example, the two predicates:

```
/a//b/c = 5
/a/b/c = 5
```

would not share the evaluation of c = 5 even though there could be states that both are evaluated. For optimal performance, subscribers should use the same style of predicates (i.e., to use descendant operators or not use descendants). In general, not using descendant operators is

faster. The sharing limitations described in this section do not prevent predicates from being evaluated properly.

9. Ideas for Extensions to this Work

One idea briefly discussed in the original YFilter paper [2] is treating simple value-based predicates as first-class citizens within the state machine. While this would be straightforward for equality, this approach becomes trickier for other comparisons. In addition, the use of simple value-based predicates at levels of the clause other than the deepest level will cause more states in the state machine and less sharing of states. In addition, the increase in states required to represent a clause would add to the insertion and deletion time of a clause. At this point we would cautiously suggest investigation into this idea. We would not suggest treating simple value-based predicates as first-class citizens in the state machine because we believe (i) more sharing occurs at the clause level and (ii) there could be a blowup in the number of states if conjunctions are treated as states.

A second idea is to more tightly integrate the simple value-based predicates into the states. Currently, an instance of the XPPath class (i.e., the class representing a compiled clause) may have simple value-based predicates at any level. However, the simple value-based predicates are currently properties of the clause (i.e., the XPPath class). If the simple value-based predicates were properties of the state, and they were sorted by the property of the document that they compare against, i.e., attribute name or text(), the pointer accesses required for the total evaluation could be reduced.

10. Conclusion

The use of YFilter concepts has been shown to improve the brokering latency and throughput for the 100X JBI. The degree of improvement will vary depending on how similar the set of predicates are. The broker using the YFilter C++ library is slightly slower for one subscriber due to the extra bookkeeping necessary for the state machine. However, the benefit of YFilter C++ can be seen with even a small number of subscribers. Speedups in broker latency of up to 15X have been experienced with the current implementation. The architecture of the 100X JBI allows scaling of throughput by adding more brokers to the system. In addition to better scaling of throughput, the implementation of YFilter C++ and subsequent integration with the 100X JBI enables improved scalability of brokering latency and therefore improved scalability of the end-to-end latency experienced by the clients. The use of the YFilterBroker also does not prevent the scaling of throughput by addition of more broker processors.

11. References

[1] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0 (Fourth Edition).* August 16, 2006. http://www.w3.org/TR/2006/REC-xml-20060816/ (accessed July 27, 2007).

[2] Diao, Yanlei, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. "Path Sharing and Predicate Evaluation for High-Performance XML Filtering." *ACM Transactions on Database Systems (TODS),* December 2003: 467-516.

[3] Diao, Yanlei, and Michael J. Franklin. "High-Performance XML Filtering: An Overview of YFilter." *IEEE Data Engineering Bulletin,* March 2003.

[4] World Wide Web Consortium. *XML Path Language.* November 16, 1999. http://www.w3.org/TR/xpath (accessed July 27, 2007).

[5] Kwong-Yan, Lok, Richard W. Linderman, and George O. Ramseyer. *100X Joint Battlespace Infosphere (JBI).* Interim Report, Rome: AFRL/IFTC, 2006.

[6] United States Air Force Scientific Advisory Board. "Report on building the Joint Battlespace Infosphere, Vol. 1: Summary." 1999.

[7] AFRL/IFSE. *Reference Implementation Quick Start Guide, Core Services Reference Implementation Version 1.2.6.* December 14, 2005. http://www.if.afrl.af.mil/programs/jbi/pages/documents/files/1.2.6/RI_Quick_Start_Guide.pdf (accessed July 27, 2007).

[8] LaValle, Steven. *Nondeterministic Finite Automata.* November 1, 2007. http://planning.cs.uiuc.edu/node558.html (accessed November 28, 2007).

[9] World Wide Web Consortium. *XQuery 1.0: An XML Query Language.* January 23, 2007. http://www.w3.org/TR/xquery/ (accessed July 24, 2007).

[10] Diao, Yanlei. *YFilter User's Manual.* http://yfilter.cs.umass.edu/html/manual/YFilter_User_Manual.html (accessed July 24, 2007).

[11] Yan, T. W., and H. Garcia-Molina. "Index structures for selective dissemination of information under boolean model." *ACM TODS* 19, no. 2 (1994): 332-334.

[12] SourceForge.net SAX Project. *About SAX*. http://sax.sourceforge.net/ (accessed July 24, 2007).

[13] Veillard, Daniel. *The XML C parser and toolkit for Gnome.* http://www.xmlsoft.org (accessed July 24, 2007).

[14] JBI Program Office, AFRL/IFSE. "RI Developer's Guide, Reference Implementation Core Services, Version 1.2.6." 2005.

[15] Soulie, Juan. *Templates.* November 16, 2007. http://www.cplusplus.com/doc/tutorial/templates.html (accessed November 28, 2007).

[16] Sandoz, Paul, Alessando Triglia, and Santiago Pericas-Geertsen. *Fast Infoset.* June, 2004. http://java.sun.com/developer/technicalArticles/xml/fastinfoset/ (accessed December 5, 2007)

[17] Free Software Foundation. *Flex - a scanner generator, Version 2.5.* March 1995. http://www.gnu.org/software/flex/manual/html_mono/flex.html (accessed July 24, 2007).

[18] Free Software Foundation. *Bison 2.3.* May 30, 2006. http://www.gnu.org/software/bison/manual/html_mono/bison.html (accessed July 24, 2007).